2

RESOURCE ALLOCATIONS & EXPERT SYSTEMS

Deliverable No.: A002

Prepared by:
JAYCOR

Prepared for:
Naval Research Laboratory
Washington, DC 20375-5000

In Response to:
Contract #N00014-85-C-2044

12 May 1988

L

The work performed to meet the requirement of this task is a continuing effort, evolving toward a general purpose reasoning tool. The idea here is to build a more powerful general expert system than the previous one ~~[1]~~. Towards that, this new Bayesian inference engine is based on the work done by Pearl and Kim ~~[2]~~. The advantages of this new inference engine over the previous one are that the representation of the knowledge is more compact and the inferencing is suitable for parallel processing.

The inference engine is written in Franz lisp on VAX machine. All the code and a typescript of how to load and use the system is attached. Keywords: Computer programming. (KR)

Accesion For		
NTIS	CRA&I	☑
DTIC	TAB	☐
Unannounced		☐
Justification		
By per ltr		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

DTIC COPY INSPECTED 6

[1] Booker L. B., An Artificial Intelligence(AI) Approach to Ship Classsification. In Intelligent Systems: Their Development and Application. Proceedings of the 24th Annual Technical Symposium, Washigton D.C. Chapter of the ACM. Gaithersburg, MD., June, 1985, p. 29-35.

[2] Kim, J. and Pearl, J., A computational Model for Combined Causal and Diagnostic Reasoning in Inference Systems, Proceedings of IJCAI-83, Los Angles, CA., August, 1983, p. 190-193.

```
cat- ls
README          driver1.o       meters/         netdif          test.l
bench           init.l          net-img.l       obs.l           typescript
cnet*           init.o          net-img.o       rtk780.c
dat/            mat.l           net.l           rtk780.o
driver1.l       mat.o           net.o           safe/
cat- lisp

2.fasl net-img
[fasl net-img]
[fasl mat.o]
[fasl init.o]
t
3.fasl driver1
[fasl driver1]
/usr/local/lib/lisp/nld -N -x -A /usr/aic/hota/bin/lisp -T f4c00 /usr/aic/hota/doc/study/cnet/examples/cat/rtk78
t
4.rtk-open
0
5.whichlevel
2
6.mfork
2
7.whichlevel
2
8.load-init

 Name of the input file to load : dat/reagan
[fasl dat/reagan.o]

 Initialize ? n

 Target node : reagan
nil
9.showbeliefs

 belief of relations_with_soviets is
                                 (0.5 0.5)
 belief of democratic_nominee is  (0.3333 0.3333 0.3333)
 belief of japanese is            (0.5 0.5)
 belief of opec_oil_prices is     (0.5 0.5)
 belief of economic_status is     (0.4119 0.3259 0.2622)
 belief of reagan is              (0.6747 0.3253)
t
10.showcons

                              pi           lambda
lk-relations_with_soviets->reagan
                         -->  (0.5 0.5)               (0.5 0.5)
lk-democratic_nominee->reagan
                         -->  (0.3333 0.3333 0.3333)  (0.3333 0.3333 0.3333)
lk-japanese->economic_status
                         -->  (0.5 0.5)               (0.5 0.5)
lk-opec_oil_prices->economic_status
                         -->  (0.5 0.5)               (0.5 0.5)
lk-economic_status->reagan
                         -->  (0.4119 0.3259 0.2622)  (0.3333 0.3333 0.3333)
t
11.change-causes

  opec_oil_pricess values are :(increased decreased)
      prior values: (1.0 1.0)
    Enter new evidence : (.3 .7)

  japaneses values are :(cooperative non_cooperative)
      prior values: (1.0 1.0)
    Enter new evidence : (.7 .3)

  democratic_nominees values are :(mondale cranston kennedy)
      prior values: (1.0 1.0 1.0)
    Enter new evidence : (.5 .4 .1)

  relations_with_sovietss values are :(friendly not_friendly)
      prior values: (1.0 1.0)
    Enter new evidence : (.4 .6)

  reelected         -> 0.7269
```

```
   fail             --> 0.2731t
 12.showbeliefs

  belief of relations_with_soviets is
                                   (0.4 0.6)
  belief of democratic_nominee is  (0.5 0.4 0.1)
  belief of japanese is            (0.7 0.3)
  belief of opec_oil_prices is     (0.3 0.7)
  belief of economic_status is     (0.5002 0.2686 0.2312)
  belief of reagan is              (0.7269 0.2731)
 t
 13.showcons

                             pi          lambda
 lk-relations_with_soviets->reagan
                         -->  (0.4 0.6)              (0.5 0.5)
 lk-democratic_nominee->reagan
                         -->  (0.5 0.4 0.1)          (0.3333 0.3333 0.3333)
 lk-japanese->economic_status
                         -->  (0.7 0.3)              (0.5 0.5)
 lk-opec_oil_prices->economic_status
                         -->  (0.3 0.7)              (0.5 0.5)
 lk-economic_status->reagan
                         -->  (0.5002 0.2686 0.2312)  (0.3333 0.3333 0.3333)
 t
 14.targetnode
 reagan
 15.bye
 (11416 . 0)
 7.bye
 77.8u 18.3s 9:57 16% 173+1177k 98+37io 658pf+0w
 cat- exit
 cat-
```

```
(eval-when (load compile eval) (load 'mat.o))
(include net.l)
```

```
(putd 'bye (getd 'exit))
(putd '+ (getd 'plus))
(putd '- (getd 'difference))
(putd '* (getd 'times))
(putd '/ (getd 'quotient))
(putd '= (getd 'equal))
(putd '** (getd 'expt))
(putd '& (getd 'and))
(putd '! (getd 'or))
(putd '~ (getd 'not))
(putd '@ (getd 'mapcar))

;;; ============= utlmac.l============
(declare (macros t))
(defmacro incf (place &optional (delta 1))
    `(setf ,place (+ ,place ,delta)))

(defmacro incf-f (place &optional (delta 1))
    `(setf ,place (+$ ,place ,delta)))

(defmacro decf (place &optional (delta 1))
    `(setf ,place (- ,place ,delta)))

(defmacro decf-f (place &optional (delta 1))
    `(setf ,place (-$ ,place ,delta)))

(defmacro for-each(fe%1 fe%2 &rest fe%rest)
    `(do ((,fe%1 (car ,fe%2) (car fe%3))
          (fe%3 (cdr ,fe%2) (cdr fe%3)))
        ((null ,fe%1) nil)
        ,(cons 'progn fe%rest)))

;(defmacro while(wh%test &rest wh%body)
;    `(do ()
;        ((not ,wh%test) nil)
;        ,(cons 'progn wh%body)))
;(defmacro while (wh%test &rest wh%body)
;    (let ((%lp (gensym)))
;        `(prog nil ,%lp
;               (or ,wh%test (return nil))
;               ,@wh%body
;               (go ,%lp))))

(defmacro ttyesno() `(yesno (read)))

(defun yesno(atm)
    (or (= atm 't)
        (= atm 'y)
        (= atm 'ye)
        (= atm 'yes)))

(defvar whichlevel 1)

(defun mfork1 macro(l)
    (list 'cond '((fork) (wait))))

(defun mfork()
    (or (mfork1) (setq whichlevel (1+ whichlevel))))

(defun flambda macro (fl)
    (list 'function (cons 'lambda (cdr fl))))

(defmacro lastcar (l_c_x)
    `(car (last ,l_c_x)))

;;; ============= flavinit.l ============
(declare (macros t))
(declare (special myhash))
(setq myhash (make-equal-hash-table))

(defflavor object+info ((insts nil)
                        (parents nil)
                        (children nil))
    ()
    :settable-instance-variables)

(defmethod (object+info :addmore)(slot val)
    (cond ((member val (symeval-in-instance self slot)))
          (t (set-in-instance self slot
                 (cons val (symeval-in-instance self slot))))))

(defun mdefflavor macro (rl)
    `(progn 'compile
            (puthash-equal (cons (quote ,(cadr rl)) '(+info))
                (make-instance 'object+info) myhash)
            (cond (,(cadddr rl)
                   (set-in-instance (gethash-equal
                                         (cons (quote ,(cadr rl)) '(+info))
                                         myhash) 'parents ,(cadddr rl))
                   (for-each x ,(cadddr rl)
                       (send (gethash-equal (cons x '(+info)) myhash)
                             ':addmore 'children ,(cadr rl)))))
            ,(cons 'defflavor (cdr rl))))

(defun mmake-instance macro(rin)
    `(progn 'compile
            (send (gethash-equal (cons ,(caddr rin) '(+info)) myhash)
                  ':addmore 'insts ,(cadr rin))
            (puthash-equal ,(cadr rin)
                ,(cons 'make-instance (cddr rin)) myhash)))


(defmacro msend (objnam slotnam &optional (slotval nil slotvalp))
    `(send (gethash-equal ,objnam myhash) ,slotnam
           ,@(if slotvalp (list slotval))))

(defmacro get-insts(obj)
    `(send (gethash-equal (cons ,obj '(+info)) myhash) ':insts))

(defun myfix (num lis)
    (do ((ans nil (cons (fix (*$ (car tmp) num)) ans))
         (tmp lis (cdr tmp)))
        ((null tmp) (nreverse ans))))

(defun myfloat (num lis)
    (setq num (float num))
    (do ((ans nil (cons (/$ (car tmp) num) ans))
         (tmp lis (cdr tmp)))
        ((null tmp) (nreverse ans))))
```

```
(eval-when (load compile eval) (load 'init.o))
;;; -=-=-=-=-= Mat.L -=-=-=-=-=-=
(declare (macros t))
(declare (special myinpport precision myhash retval))
; -=-=-=-=-= remove-duplicates -=-=-=-=
(defun remove-duplicates (lst &optional (from-end nil))
    (do ((ans (cond (from-end (cdr lst)) (t (cdr (reverse lst)))))
         (mark (cond (from-end (list (car lst))) (t (last lst)))
               (cond (ans (cons (car ans) mark)) (t mark))))
        ((null ans) (cond (from-end (reverse mark))
                          (t mark)))
        (setq ans (remove (car mark) ans))))


; -=-=-=-=-= normalizing -=-=-=-=-=
; (norm '(.2 .3)) => (.4 .6)
(defun norm(lst)
    (let ((sum (apply '+$ lst)))
         (declare (flonum sum))
         (cond ((zerop sum)
                (listsomany (length lst) (/$ 1.0 (length lst))))
               (t (do ((ans1 nil
                             (cons (/$ (car lst1) sum) ans1))
                       (lst1 lst (cdr lst1)))
                      ((null lst1) (nreverse ans1)))))))


; -=-=-=-=-= outer product -=-=-=-=-=
;(outerpro2 '(1 2) '(.2 .3)) => (.2 .3 .4 .6)
(defun outerpro2(l1 l2)
    (do ((ans nil
              (append
                    (do ((xelt tmp2 (cdr xelt))
                         (tmplcar (car tmpl))
                         (ans1 nil (cons (*$ tmplcar (car xelt)) ans1)))
                        ((null xelt) ans1))
                    ans))
         (tmpl (reverse l1) (cdr tmpl))
         (tmp2 (reverse l2)))
        ((null tmpl) ans)))
;(outerpro '((1 2) (.2 .3) (.4 .5))) => (.08 .1 .12 .15 .16 .2 .24 .32)
(defun outerpro(l)
    (do ((ans (lastcar l)
              (do ((xelt (car nav) (cdr xelt))
                   (ans1 nil))
                  ((null xelt) (nreverse ans1))
                  (do ((yelt ans (cdr yelt))
                       (xeltcar (car xelt)))
                      ((null yelt))
                      (setq ans1 (cons (*$ xeltcar (car yelt)) ans1)))))
         (nav (cdr (reverse l)) (cdr nav)))
        ((null nav) ans)))

; (outerpromany '(((.2 .3) (.4 .5)) ((.1 .2) (.3 .4))))
;       => ((.08 .1 .12 .15) (.03 .04 .06 .08))
(defun outerpromany (lis)
    (do ((ans nil (cons (outerpro (car tmp)) ans))
         (tmp lis (cdr tmp)))
        ((null tmp) (nreverse ans))))

(eval-when (load eval) (putd 'aref (getd 'funcall)))
(defsetf aref (e v) `(funcall ,(cadr e) ,v ,@(cddr e)))
(defvar *maklis-tmp* (*array nil t 10))
(setf (aref *maklis-tmp* 0) nil)
(loop for i from 1 to 9
    do (setf (aref *maklis-tmp* i)
             (append (aref *maklis-tmp* (1- i))
                     (list i))))
; (maklis 3) => (1 2 3)
(defun maklis(n)
    (declare (fixnum n))
    (cond ((< n 10) (aref *maklis-tmp* n))
          (t (append (maklis (1- n)) (list n)))))


; (listsomany 4 A) => (A A A A)
(defun listsomany (l e)
    (declare (fixnum l))
    (cond ((< l 1) nil)
          ((= l 1) (list e))
          (t (cons e (listsomany (- l 1) e)))))

; (lisdiv '(1 2 3 4 5) 3) => ((1 2 3) (4 5))
(defun lisdiv (lis n1)
    (declare (fixnum n1))
    (let ((x1))
         (while (> n1 0)
                (setq x1 (cons (car lis) x1)
                      lis (cdr lis)
                      n1 (1- n1)))
         (list (nreverse x1) lis)))

(defun lisdivrec (lis n1)
    (do ((ans nil
              (cons (do ((ans1 nil (cons (car tmp) ans1))
                         (junk nil (setq tmp (cdr tmp)))
                         (n2 n1 (1- n2)))
                        ((zerop n2) (nreverse ans1)))
                    ans))
         (tmp lis))
        ((null tmp) (nreverse ans))))


; -=-=-=-=-= term product -=-=-=-=-=
; (termpro2 '(1 2) '(3 4)) => (3 8)
(defun termpro2(l1 l2)
    (do ((ans nil (cons (*$ (car tmp1) (car tmp2)) ans))
         (tmp1 l1 (cdr tmp1))
         (tmp2 l2 (cdr tmp2)))
        ((null tmp1) (nreverse ans))))

; (termpro '((1 2) (3 4) (5 6))) => (15 48)
(defun termpro(l)
    (do ((ans (car l)
              (do ((ans1 nil (cons (*$ (car tmp1) (car tmp2)) ans1))
                   (tmp1 ans (cdr tmp1))
                   (tmp2 (car nav) (cdr tmp2)))
                  ((null tmp1) (nreverse ans1))))
         (nav (cdr l) (cdr nav)))
        ((null nav) ans)))

; (invert '((1 2 3) (4 5 6))) => ((1 4) (2 5) (3 6))
(defun invert(ppro)
    (let ((rppro (reverse ppro)))
         (do ((ans (mapcar 'list (car rppro))
                   (do ((ans1 nil
                              (cons (cons (car tmp1) (car tmp2))
                                    ans1))
                        (tmp1 (car nav) (cdr tmp1))
                        (tmp2 ans (cdr tmp2)))
                       ((null tmp1) (nreverse ans1))))
              (nav (cdr rppro) (cdr nav)))
             ((null nav) ans))))

(defun arrange (ls ord which thispi)
    (setq ord (lisdiv ord (1- which)))
    (do ((ans nil (cons (/$ (apply '+$ (car tmp)) (car tmppi)) ans))
         (tmppi thispi (cdr tmppi))
         (tmp (let ((pr (apply '*  (car ord)))
                    (tr (caadr ord))
                    (nr (apply '* (cdadr ord))))
                   (cond ((= nr 1) (invert (lisdivrec ls tr)))
                         ((= pr 1) (lisdivrec ls nr))
                         (t (invert1 (lisdivrec (lisdivrec ls nr) tr)))))
              (cdr tmp)))
        ((null tmp) (nreverse ans))))

(defun arrange1 (ls ord which thispi)
    (setq ord (lisdiv ord (1- which)))
    (do ((ans nil (cons (/$ (car tmp) (car tmppi)) ans))
         (tmppi thispi (cdr tmppi))
         (tmp (let ((pr (apply '*  (car ord)))
                    (tr (caadr ord))
                    (nr (apply '* (cdadr ord))))
                   (cond ((= nr 1) (invertadd ls tr))
                         ((= pr 1) (addsomany ls nr))
                        (t (invertadd (addsomany ls nr) tr))))
                (cdr tmp)))
          ((null tmp) (nreverse ans))))

(defun addsomany (lis nl)
    (do ((ans nil
              (cons (do ((ansl 0 (+$ (car tmp) ansl))
                         (junk nil (setq tmp (cdr tmp)))
                         (n2 nl (1- n2)))
                        ((zerop n2) ansl))
                    ans))
         (tmp lis))
        ((null tmp) (nreverse ans))))

(defun invertadd (lis nl)
    (let ((temp (lisdiv lis nl)))
         (do ((ans (car temp)
                   (do ((ansl nil
                              (cons (+$ (car tmp) (car ans2)) ansl))
                        (ans2 ans (cdr ans2))
                        (junk nil (setq tmp (cdr tmp)))
                        (n2 nl (1- n2)))
                       ((zerop n2) (nreverse ansl))))
              (tmp (cadr temp)))
             ((null tmp) ans))))


(defun invertl(ppro)
    (let ((rppro (reverse ppro)))
         (do ((ans (car rppro)
                   (do ((ansl nil
                              (cons (append (car tmpl) (car tmp2))
                                    ansl))
                        (tmpl (car nav) (cdr tmpl))
                        (tmp2 ans (cdr tmp2)))
                       ((null tmpl) (nreverse ansl))))
              (nav (cdr rppro) (cdr nav)))
             ((null nav) ans))))

; (matdiv '(1 2 3 4) '(.5 1.0 2.0 3.0)) => (2.0 2.0 1.5 1.333)
(defun matdiv (bel y)
    (declare (flonum y))
    (do ((ans nil
              (cons (/$ (car tmpl) (car tmp2))
                    ans))
         (tmpl bel (cdr tmpl))
         (tmp2 y (cdr tmp2)))
        ((null tmpl) (nreverse ans))))


(defun findcp(ppro)
    (do ((ans nil (cons (myfix 10000 (norm (car tmpl))) ans))
         (tmpl (do ((ansl (car ppro)
                          (do ((ans2 nil
                                     (cons (outerpro2 (car tmp3)
                                                      (car tmp4)) ans2))
                               (tmp3 ansl (cdr tmp3))
                               (tmp4 (car tmp2) (cdr tmp4)))
                              ((null tmp3) (nreverse ans2))))
                    (tmp2 (cdr ppro) (cdr tmp2)))
                   ((null tmp2) (invert ansl)))
               (cdr tmpl)))
        ((null tmpl) (nreverse ans))))

(defun myequal (ls1 ls2)
    (do ((tmpl ls1 (cdr tmpl))
         (tmp2 ls2 (cdr tmp2)))
        ((or (null tmpl)
             (null tmp2)
             (not (mydiff (car tmpl) (car tmp2))))
         (and (null tmpl) (null tmp2)))))

(defun mydiff(x y)
    (declare (flonum x y))
    (and (< (-$ y x) precision)
         (< (-$ x y) precision)))

(defun msendal(namh slotset slotget val)
        (send namh slotset
              (nreverse (cons val (nreverse (send namh slotget))))))

    (defun mspsend (lnkh slot val)
        (cond ((myequal (send lnkh slot) val)
               (cond ((equal slot ':pi) (send lnkh ':set-pi val) nil)
                     ((equal slot ':lambda) (send lnkh ':set-lambda val) nil)))
              ((equal slot ':pi)
               (send lnkh ':set-pi val) (send lnkh ':bnode))
              ((equal slot ':lambda)
               (send lnkh ':set-lambda val) (send lnkh ':tnode))))
```

```
; -=-=-=-=-=-=net.l -=-=-=-=
; -=-=-=   At a node -=-=

(declare (macros t))
(declare (special  myinpport precision myhash
(defvar tobe-updated nil)
(defvar all-nodes nil)
(defvar all-links nil)
(defvar all-nodesh nil)
(defvar all-linksh nil)

(defmacro mreset()
    '(progn()
           (setq myhash (make-equal-hash-table))
           (mdefflavor node
               ((values '(True False))
                (rank nil)
                (tlinks nil)
                (blinks nil)
                (prior nil)
                (parranks nil)
                (parprobs nil)
                (condprol nil)
                (condpro2 nil)
                (belief nil))
               ()
               :settable-instance-variables)

           (mdefflavor link
               ((tnode nil)
                (bnode nil)
                (indpro nil)
                (lambda nil)
                (pi nil))
               ()
               :settable-instance-variables)))

;(mreset)                                  ; initializing
    (setq myhash (make-equal-hash-table))
            (mdefflavor node
                ((values '(True False))
                 (rank nil)
                 (tlinks nil)
                 (blinks nil)
                 (prior nil)
                 (parranks nil)
                 (parprobs nil)
                 (condprol nil)
                 (condpro2 nil)
                 (belief nil))
                ()
                :settable-instance-variables)

            (mdefflavor link
                ((tnode nil)
                 (bnode nil)
                 (indpro nil)
                 (lambda nil)
                 (pi nil))
                ()
                :settable-instance-variables)


(defun mdescribe(nnam)
    (describe (gethash-equal nnam myhash)))

(defun shownodes() (mapc 'mdescribe all-nodes))
(defun showlinks() (mapc 'mdescribe all-links))
(defun shownetwork() (mapc 'mdescribe (append all-nodes all-links)))

;;; ------------------------- updateall ------------------------
;(defun updateall-dn(&optional (what-node nil))
;     (let ((tobe-updated (cond (what-node (list what-node))
;                               (t all-nodes)))
;          (current))
;         (while tobe-updated
;                (setq current (gethash-equal (car tobe-updated) myhash)
;                      tobe-updated (cdr tobe-updated))
;               (send current ':update))))
;
;(defun updateall-bn(&optional (what-node nil))
;    (let ((tobe-updated (cond (what-node (list what-node))
;                              (t all-nodes)))
;          (current))
;         (while tobe-updated
;                (setq current (gethash-equal (car (last tobe-updated))
;                                 myhash)
;                      tobe-updated (reverse (cdr (reverse tobe-updated))))
;                (send current ':update))))
;
;(defun updateall-dp(&optional (what-node nil))
;    (let ((tobe-updated (cond (what-node (list what-node))
;                              (t all-nodes)))
;          (current))
;         (while tobe-updated
;                (setq tobe-updated (remove-duplicates tobe-updated)
;                      current (gethash-equal (car tobe-updated) myhash)
;                      tobe-updated (cdr tobe-updated))
;                (send current ':update))))
;
;(defun updateall-dpe(&optional (what-node nil))
;    (let ((tobe-updated (cond (what-node (list what-node))
;                              (t all-nodes)))
;          (current))
;         (while tobe-updated
;                (setq tobe-updated (remove-duplicates tobe-updated t)
;                      current (gethash-equal (car tobe-updated) myhash)
;                      tobe-updated (cdr tobe-updated))
;                (send current ':update))))
;
(defun updateall-br(&optional (what-node nil))
    (let ((tobe-updated (cond (what-node (list what-node))
                              (t all-nodes)))
          (current))
         (while tobe-updated
                (setq tobe-updated (remove-duplicates tobe-updated)
                      current (gethash-equal (car (last tobe-updated))
                                 myhash)
                      tobe-updated (reverse (cdr (reverse tobe-updated))))
                (send current ':update))))

;(defun updateall-bre(&optional (what-node nil))
;    (let ((tobe-updated (cond (what-node (list what-node))
;                              (t all-nodes)))
;          (current))
;         (while tobe-updated
;                (setq tobe-updated (remove-duplicates tobe-updated t)
;                      current (gethash-equal (car (last tobe-updated))
;                                 myhash)
;                      tobe-updated (reverse (cdr (reverse tobe-updated))))
;                (send current ':update))))

;;;  ------------ end updating --------------------
(defun showbeliefs(&optional (ofwhat all-nodes))
    (cond ((atom ofwhat) (msg (N 1) "belief of " ofwhat " is "
                              (C 35) (msend ofwhat ':belief)))
          (t (mapc (flambda(x)
                          (msg (N 1) " belief of " x " is "
                               (C 35) (msend x ':belief)))
                   ofwhat))))

(defun showcons(&optional (ofwhat all-links))
    (msg (N 1) (B 30) "pi" (B 10) "lambda")
    (cond ((atom ofwhat)
           (msg (N 1)  ofwhat (C 25) " -->  " (msend ofwhat ':pi) (C 55)
                (msend ofwhat ':lambda)))
          (t (mapc (flambda(x)
                          (msg (N 1)  x (C 25) " -->  " (msend x ':pi) (C 55)
                               (msend x ':lambda)))
                   ofwhat))))

(defun showlambdas(&optional (ofwhat all-links))
    (cond ((atom ofwhat) (msg (N 1) "lambda of " ofwhat " is "
                              (C 35) (msend ofwhat ':lambda)))
          (t (mapc (flambda(x)
                          (msg (N 1) " lambda of " x " is "
                          (C 35) (msend x ':lambda)))
              ofwhat))))

(defun showpis(&optional (ofwhat all-links))
    (cond ((atom ofwhat) (msg (N 1) "pi of " ofwhat " is "
                             (C 35) (msend ofwhat ':pi)))
          (t (mapc (flambda(x)
                          (msg (N 1) " pi of " x " is "
                               (C 35) (msend x ':pi)))
              ofwhat))))

(defmethod (node :getallpis) ()
    (cond (tlinks (do ((ans nil
                         (cons (send (gethash-equal
                                         (car tmpl) myhash) ':pi)
                               ans))
                     (tmpl tlinks (cdr tmpl)))
                    ((null tmpl) (nreverse ans))))
          (t (list prior))))

(defmethod (node :getalllambdas) ()
    (cond (blinks (do ((ans nil
                         (cons (send (gethash-equal
                                         (car tmpl) myhash)
                                     ':lambda)
                               ans))
                     (tmpl blinks (cdr tmpl)))
                    ((null tmpl) (nreverse ans))))
          (t (list prior))))


(defmethod (node :update) ()
    (let* ((ptlinksln (length tlinks))
           (pblinksln (length blinks))
           (allpis (send self ':getallpis))
           (alllambdas (send self ':getalllambdas))
           (piout (outerpro allpis))
           (prelambda (termpro alllambdas))
           (bel) (prepi) (contlam))

          (cond ((= ptlinksln 0) ; *** no top links ***
                 (setq bel (termpro2 prelambda piout)))
                (t
                  (setq bel
                        (termpro2
                        (myfloat 100000000
                            (do ((ans nil
                                     (cons (do ((ansl 0
                                                          (+ ansl
                                                             (* (car mtl)
                                                                (car mt2))))
                                                 (mtl (car templ)
                                                      (cdr mtl))
                                                 (mt2 temp2 (cdr mt2)))
                                                ((null mtl) ansl))
                                           ans))
                                 (templ condpro2 (cdr templ))
                                 (temp2 (myfix 10000 piout)))
                                ((null templ) (nreverse ans))))
                        prelambda)

                        contlam
                        (myfloat 100000000
                            (do ((ans nil
                                     (cons (do ((ansl 0 (+ ansl
                                                              (* (car mtl)
                                                                 (car mt2))))
                                                 (mtl (car templ) (cdr mtl))
                                                 (mt2 temp2 (cdr mt2)))
                                                ((null mtl) ansl))
                                           ans))
                                 (templ condprol (cdr templ))
                                 (temp2 (myfix 10000 prelambda)))
                                ((null templ) (nreverse ans)))))))

         (send self ':set-belief (norm bel)); update belief

         (do ((templ blinks (cdr templ)); update pis
                (temp2 alllambdas (cdr temp2)))
               ((null templ))
               (let ((temp3 (mspsend (gethash-equal (car templ)
                                                    myhash)
                                     ':pi (matdiv bel (car temp2)))))
                    (cond (temp3 (push temp3 tobe-updated)))))

          (cond ((= ptlinksln 1); update lambdas
                 (let ((temp (mspsend (gethash-equal (car tlinks) myhash)
                                      ':lambda contlam)))
                      (cond (temp (push temp tobe-updated)))))
                ((> ptlinksln 1)
                 (do ((templ tlinks (cdr templ))
                      (temp2 (maklis (length parranks)) (cdr temp2))
                      (temp3 (termpro2 contlam piout);)
                     ((null templ))
                     (let* ((tclh (gethash-equal (car templ)
                                                 myhash))
                            (temp (mspsend tclh ':lambda
                                           (arrange temp3
                                              parranks (car temp2)
                                              (send tclh ':pi)))))
                           (cond (temp (push temp tobe-updated))))))))

; -=*-=*-=*
(defun cnet()
    (msg (N 1) (ptime) (N 1))
    (setq precision (/$ 1.0 (expt 10 4)))
    (setq float-format "%.4g")
    (msg (N 1) " Name of the input file to load : ")
    (load  (read))
    (setq all-nodes (get-insts 'node)
          all-links (get-insts 'link)
          all-nodesh (do ((ans nil
                              (cons (gethash-equal (car tmp) myhash)
                                    ans))
                          (tmp all-nodes (cdr tmp)))
                         ((null tmp) (nreverse ans)))
          all-linksh (do ((ans nil
                              (cons (gethash-equal (car tmp) myhash)
                                    ans))
                          (tmp all-links (cdr tmp)))
                         ((null tmp) (nreverse ans))))

    (msg (N 1) " Initialize ? ")
    (and (ttyesno) (init-net) (down-load-all))
    (msg (N 1) (ptime) (N 1))
    (updateall-br)
    (msg (N 1) (ptime) (N 1)))


(defun init-net()
    ; add tlinks and blinks to nodes & check cardinality of indpro
    (for-each xh all-nodesh
        (let ((pri (send xh ':prior)))
             (send xh ':set-rank (length (send xh ':values)))
             (cond (pri (send xh ':set-prior (norm pri)))
                   (t (send xh ':set-prior
                           (listsomany (send xh ':rank) 1.0))))
             (send xh ':set-belief (send xh ':prior))))

    (for-each x all-links
        (let ((xh (gethash-equal x myhash)))
             (msendal (gethash-equal (send xh ':tnode) myhash)
                 ':set-blinks ':blinks x)
             (msendal (gethash-equal (send xh ':bnode) myhash)
                 ':set-tlinks ':tlinks x)))


    ; expand all nodes
    (for-each xh all-nodesh
        (let ((tlk (send xh ':tlinks)))
             (cond (tlk (do ((parpro nil
                                     (cons (send (gethash-equal (car tlks)
                                                         myhash) ':indpro)
                                           parpro))
                             (parrnk nil
                                        (cons (msend (msend (car tlks) ':tnode)
                                                            ':rank)
                                              parrnk))
                                  (tlks (reverse tlk) (cdr tlks)))
                                 ((null tlks)
                                  (send xh ':set-parprobs parpro)
                                  (send xh ':set-parranks parrnk)
                                  (send xh ':set-condpro1
                                        (findcp parpro))
                                  (send xh ':set-condpro2
                                        (invert (send xh ':condpro1)))
                                 ))))))

    ; expand all links
    (for-each xh all-linksh
        (send xh ':set-pi (send (gethash-equal (send xh ':tnode)
                                    myhash) ':prior))
        (send xh ':set-lambda
              (do ((ans nil
                        (cons (do ((ans1 0
                                        (+$ ans1 (*$ (car mt1)
                                                     (car mt2))))
                                   (mt1 (car tem1) (cdr mt1))
                                   (mt2 tem2 (cdr mt2)))
                                  ((null mt1) ans1))
                              ans))
                   (tem1 (invert (send xh ':indpro)) (cdr tem1))
                   (tem2 (send (gethash-equal (send xh ':bnode) myhash)
                               ':prior)))
                  ((null tem1) (nreverse ans)))))
    t)

(defun d-l-node (x)
    (let ((xh (gethash-equal x myhash)))
         (msg (N 1) "(mmake-instance '" x " 'node"
              (N 1) " ':values      '" (send xh ':values)
              (N 1) " ':rank        '" (send xh ':rank)
              (N 1) " ':tlinks      '" (send xh ':tlinks)
              (N 1) " ':blinks      '" (send xh ':blinks)
              (N 1) " ':prior       '" (send xh ':prior)
              (N 1) " ':parranks    '" (send xh ':parranks)
              (N 1) " ':parprobs    '" (send xh ':parprobs)
              (N 1) " ':condpro1    '" (send xh ':condpro1)
              (N 1) " ':condpro2    '" (send xh ':condpro2)
              (N 1) " ':belief      '" (send xh ':belief) ")"
              (N 2))))
(defun d-l-link (x)
    (let ((xh (gethash-equal x myhash)))
         (msg (N 1) "(mmake-instance '" x " 'link"
              (N 1) " ':tnode       '" (send xh ':tnode)
              (N 1) " ':bnode       '" (send xh ':bnode)
              (N 1) " ':indpro      '" (send xh ':indpro)
              (N 1) " ':lambda      '" (send xh ':lambda)
              (N 1) " ':pi          '" (send xh ':pi) ")"
              (N 2))))
(defun down-load-all()
    (msg (N 1) "; =========NODES=======" (N 1))
    (mapc 'd-l-node (get-insts 'node))
    (msg (N 1) "; =========LINKS=======" (N 1))
    (mapc 'd-l-link (get-insts 'link)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Lowercase: Yes; Base: 10; Fonts:CPTFONT,CPTFONTI,[illegible]
;;; ------------ graphic interface ----------------
(declare (macros t))
;(eval-when (load compile) (load '780load.o))
(cfasl '~hota/doc/study/cnet/examples/cat/rtk780.o '_device_open 'rtk-open
    "integer-function")
(getaddress '_device_close   'rtk-close  "integer-function")
(getaddress '_device_set_artype    'set-artype   "integer-function")
(getaddress '_device_whatalu    'whatalu   "integer-function")
(getaddress '_device_line    'rtk-line   "integer-function")
(getaddress '_device_rect    'rtk-rect   "integer-function")
(getaddress '_device_rect_border 'rtk-rect-border "integer-function")
(getaddress '_device_text    'rtk-text   "integer-function")
(getaddress '_device_flipcol 'flipcol "integer-function")
(getaddress '_device_erase   'rtk-erase  "integer-function")

(eval-when (load) (rtk-open 1))
(defun graph-pane-width() 1280)
(defun graph-pane-height() 1024)
(defun draw-myline (x0 y0 x1 y1 &optional (color 1) (artype 'tv:alu-xor))
  (set-artype (cond ((= artype 'tv:alu-xor) #x2)
                    (t #x0)))
  (rtk-line x0 y0 x1 y1 color))
(defun draw-mystring (iname x1 y1 x2 y2
                            &optional (fcolor 1) (bcolor 0) (dx 8) (dy 12))
  (rtk-text x1 y1 fcolor bcolor iname dx dy))
(defun draw-mygray (&rest rest))
(defun draw-myrectangle (x0 y0 x1 y1 &optional (color 1) (artype 'tv:alu-xor))
  (set-artype (cond ((= artype 'tv:alu-xor) #x2)
                    (t #x0)))
  (rtk-rect x1 y1 (+ x0 x1) (+ y0 y1) color))
(defun draw-myrectangle-border (x0 y0 x1 y1 color)
    (rtk-rect-border x0 y0 x1 y1 color))
(defun clear-graph-pane(&rest rest)
  (rtk-erase))
(defun draw-bitblt(&rest rest))
(defmacro tr-graph()
    '(trace graph-pane-width
           graph-pane-height
           draw-myline
           draw-mystring
           draw-mygray
           draw-myrectangle
           clear-graph-pane
           draw-bitblt))
(defun testcol()
  (do ((y1 0 (+ y1 64))
       (clnum 0))
       ((= y1 1024) t)
    (do ((x1 0 (+ x1 80)))
        ((= x1 1280) t)
      (rtk-rect x1 y1 (+ x1 79) (+ y1 63) clnum)
      (rtk-rect-border x1 y1 (+ x1 79) (+ y1 63) 0)
      (setf clnum (1+ clnum)))))

(defun tcls(&optional (fg 1))
  (do ((y1 0 (+ y1 128))
       (clnum 0))
       ((= y1 1024) t)
    (do ((x1 0 (+ x1 80)))
        ((= x1 1280) t)
      (rtk-rect x1 y1 (+ x1 79) (+ y1 127) clnum)
      (rtk-rect-border x1 y1 (+ x1 79) (+ y1 127) 0)
      (rtk-text (+ x1 y1 0 124 (get_pname (concat clnum ""))))
      (setf clnum (next-clnum clnum fg)))))

(defun next-clnum (pcol inc)
    (let ((tmp (+ pcol inc)))
        (cond ((< tmp 128) tmp)
              (t (1+ (mod tmp inc))))))
```

```
#include <sys/ioctl.h>
#include <sys/rtk.h>
#include <stdio.h>
#include <signal.h>

static char buff[2048];
static short set = 0x800;
static int ramtek;
static char alutype;
/* Black Red Orange Yellow Green Blue Aqua White & 8 more */
char table[1024] = {            /* Actually should be "table[256][4]"... */
                0,0,0,0,
                0,0,255,0,
                0,128,255,0,
                0,255,255,0,
                0,255,0,0,
                255,0,0,0,
                255,255,0,0,
                255,255,255,0,
                0,0,0,1,
                0,0,255,1,
                0,128,255,1,
                0,255,255,1,
                0,255,0,1,
                255,0,0,1,
                255,255,0,1,
                255,255,255,1
};
char table1[1024] = {
          0,    0,    0,  0,
          0,    0,   63,  0,
          0,    0,  127,  0,
          0,    0,  191,  0,
          0,    0,  255,  0,

          0,   63,    0,  0,
          0,   63,   63,  0,
          0,   63,  127,  0,
          0,   63,  191,  0,
          0,   63,  255,  0,

          0,  127,    0,  0,
          0,  127,   63,  0,
          0,  127,  127,  0,
          0,  127,  191,  0,
          0,  127,  255,  0,

          0,  191,    0,  0,
          0,  191,   63,  0,
          0,  191,  127,  0,
          0,  191,  191,  0,
          0,  191,  255,  0,

          0,  255,    0,  0,
          0,  255,   63,  0,
          0,  255,  127,  0,
          0,  255,  191,  0,
          0,  255,  255,  0,

         63,    0,    0,  0,
         63,    0,   63,  0,
         63,    0,  127,  0,
         63,    0,  191,  0,
         63,    0,  255,  0,

         63,   63,    0,  0,
         63,   63,   63,  0,
         63,   63,  127,  0,
         63,   63,  191,  0,
         63,   63,  255,  0,

         63,  127,    0,  0,
         63,  127,   63,  0,
         63,  127,  127,  0,
         63,  127,  191,  0,
         63,  127,  255,  0,

         63,  191,    0,  0,
    63,    191,  63,   0,
    63,    191, 127,   0,
    63,    191, 191,   0,
    63,    191, 255,   0,

    63,    255,   0,   0,
    63,    255,  63,   0,
    63,    255, 127,   0,
    63,    255, 191,   0,
    63,    255, 255,   0,

   127,      0,   0,   0,
   127,      0,  63,   0,
   127,      0, 127,   0,
   127,      0, 191,   0,
   127,      0, 255,   0,

   127,     63,   0,   0,
   127,     63,  63,   0,
   127,     63, 127,   0,
   127,     63, 191,   0,
   127,     63, 255,   0,

   127,    127,   0,   0,
   127,    127,  63,   0,
   127,    127, 127,   0,
   127,    127, 191,   0,
   127,    127, 255,   0,

   127,    191,   0,   0,
   127,    191,  63,   0,
   127,    191, 127,   0,
   127,    191, 191,   0,
   127,    191, 255,   0,

   127,    255,   0,   0,
   127,    255,  63,   0,
   127,    255, 127,   0,
   127,    255, 191,   0,
   127,    255, 255,   0,

   191,      0,   0,   0,
   191,      0,  63,   0,
   191,      0, 127,   0,
   191,      0, 191,   0,
   191,      0, 255,   0,

   191,     63,   0,   0,
   191,     63,  63,   0,
   191,     63, 127,   0,
   191,     63, 191,   0,
   191,     63, 255,   0,

   191,    127,   0,   0,
   191,    127,  63,   0,
   191,    127, 127,   0,
   191,    127, 191,   0,
   191,    127, 255,   0,

   191,    191,   0,   0,
   191,    191,  63,   0,
   191,    191, 127,   0,
   191,    191, 191,   0,
   191,    191, 255,   0,

   191,    255,   0,   0,
   191,    255,  63,   0,
   191,    255, 127,   0,
   191,    255, 191,   0,
   191,    255, 255,   0,

   255,      0,   0,   0,
   255,      0,  63,   0,
   255,      0, 127,   0,
   255,      0, 191,   0,
   255,      0, 255,   0,

   255,     63,   0,   0,
        255,   63,   63,   0,
        255,   63,  127,   0,
        255,   63,  191,   0,
        255,   63,  255,   0,

        255,  127,    0,   0,
        255,  127,   63,   0,
        255,  127,  127,   0,
        255,  127,  191,   0,
        255,  127,  255,   0,

        255,  191,    0,   0,
        255,  191,   63,   0,
        255,  191,  127,   0,
        255,  191,  191,   0,
        255,  191,  255,   0,

        255,  255,    0,   0,
        255,  255,   63,   0,
        255,  255,  127,   0,
        255,  255,  191,   0,
        255,  255,  255,   0,

         32,   32,   32,   0,
         95,   95,   95,   0,
        159,  159,  159,   0,

          0,    0,    0,   1,
          0,    0,   63,   1,
          0,    0,  127,   1,
          0,    0,  191,   1,
          0,    0,  255,   1,

          0,   63,    0,   1,
          0,   63,   63,   1,
          0,   63,  127,   1,
          0,   63,  191,   1,
          0,   63,  255,   1,

          0,  127,    0,   1,
          0,  127,   63,   1,
          0,  127,  127,   1,
          0,  127,  191,   1,
          0,  127,  255,   1,

          0,  191,    0,   1,
          0,  191,   63,   1,
          0,  191,  127,   1,
          0,  191,  191,   1,
          0,  191,  255,   1,

          0,  255,    0,   1,
          0,  255,   63,   1,
          0,  255,  127,   1,
          0,  255,  191,   1,
          0,  255,  255,   1,

         63,    0,    0,   1,
         63,    0,   63,   1,
         63,    0,  127,   1,
         63,    0,  191,   1,
         63,    0,  255,   1,

         63,   63,    0,   1,
         63,   63,   63,   1,
         63,   63,  127,   1,
         63,   63,  191,   1,
         63,   63,  255,   1,

         63,  127,    0,   1,
         63,  127,   63,   1,
         63,  127,  127,   1,
         63,  127,  191,   1,
         63,  127,  255,   1,

         63,  191,    0,   1,
         63,  191,   63,   1,
         63,  191,  127,   1,
         63,    191, 191,   1,
         63,    191, 255,   1,

         63,    255,   0,   1,
         63,    255,  63,   1,
         63,    255, 127,   1,
         63,    255, 191,   1,
         63,    255, 255,   1,

        127,      0,   0,   1,
        127,      0,  63,   1,
        127,      0, 127,   1,
        127,      0, 191,   1,
        127,      0, 255,   1,

        127,     63,   0,   1,
        127,     63,  63,   1,
        127,     63, 127,   1,
        127,     63, 191,   1,
        127,     63, 255,   1,

        127,    127,   0,   1,
        127,    127,  63,   1,
        127,    127, 127,   1,
        127,    127, 191,   1,
        127,    127, 255,   1,

        127,    191,   0,   1,
        127,    191,  63,   1,
        127,    191, 127,   1,
        127,    191, 191,   1,
        127,    191, 255,   1,

        127,    255,   0,   1,
        127,    255,  63,   1,
        127,    255, 127,   1,
        127,    255, 191,   1,
        127,    255, 255,   1,

        191,      0,   0,   1,
        191,      0,  63,   1,
        191,      0, 127,   1,
        191,      0, 191,   1,
        191,      0, 255,   1,

        191,     63,   0,   1,
        191,     63,  63,   1,
        191,     63, 127,   1,
        191,     63, 191,   1,
        191,     63, 255,   1,

        191,    127,   0,   1,
        191,    127,  63,   1,
        191,    127, 127,   1,
        191,    127, 191,   1,
        191,    127, 255,   1,

        191,    191,   0,   1,
        191,    191,  63,   1,
        191,    191, 127,   1,
        191,    191, 191,   1,
        191,    191, 255,   1,

        191,    255,   0,   1,
        191,    255,  63,   1,
        191,    255, 127,   1,
        191,    255, 191,   1,
        191,    255, 255,   1,

        255,      0,   0,   1,
        255,      0,  63,   1,
        255,      0, 127,   1,
        255,      0, 191,   1,
        255,      0, 255,   1,

        255,     63,   0,   1,
        255,     63,  63,   1,
        255,     63, 127,   1,
	255,   63,  191,  1,
	255,   63,  255,  1,

	255,   127,   0,  1,
	255,   127,  63,  1,
	255,   127, 127,  1,
	255,   127, 191,  1,
	255,   127, 255,  1,

	255,   191,   0,  1,
	255,   191,  63,  1,
	255,   191, 127,  1,
	255,   191, 191,  1,
	255,   191, 255,  1,

	255,   255,   0,  1,
	255,   255,  63,  1,
	255,   255, 127,  1,
	255,   255, 191,  1,
	255,   255, 255,  1,

	 32,    32,  32,  0,
	 95,    95,  95,  0,
	159,   159, 159,  0
	  };

device_open(orientation)
int orientation;
{
	if((ramtek = open("/dev/rtk", 2)) < 0) return(-1);
	r_wtab(table1, 0, 0, 256, 0);
	alutype = 0x00;
	buff[0] = orientation & 0x01;
	buff[1] = 0x27;
	if(write(ramtek, buff, 2) != 2) return(-1);
	return(0);
}

device_close()
{
	close(ramtek);
	return(0);
}

device_set_artype(x)
int *x;
{
	switch(*x) {
	case 0: alutype = 0x00; break;
	case 1: alutype = 0x01; break;
	case 2: alutype = 0x02; break;
	case 3: alutype = 0x03; break;
	case 4: alutype = 0x04; break;
	case 5: alutype = 0x05; break;
	case 6: alutype = 0x06; break;
	case 7: alutype = 0x07; break;
	case 8: alutype = 0x08; break;
	case 9: alutype = 0x09; break;
	case 10: alutype = 0x0a; break;
	case 11: alutype = 0x0b; break;
	case 12: alutype = 0x0c; break;
	default: alutype = 0x00; break;
	}

}
device_whatalu()
{
    return (alutype - 0x00);
}
device_line(x0, y0, x1, y1, color)
int *x0, *y0, *x1, *y1, *color;
{
	register int rx0, ry0, rx1, ry1;

	rx0 = *x0;
	ry0 = *y0;
	rx1 = *x1;
	ry1 = *y1;
```

```
        buff[0] = 0x03; buff[1] = 0x0E;
        buff[2] = 0x02; buff[3] = 0x88;
        buff[4] = *color & 0xFF; buff[5] = 0x00;
        buff[6] = alutype; buff[7] = 0x00;
        buff[8] = rx0 & 0xFF; buff[9] = (rx0 >> 8) & 0x07;
        buff[10] = ry0 & 0xFF; buff[11] = (ry0 >> 8) & 0x03;
        buff[12] = 0x04; buff[13] = 0x00;
        buff[14] = rx1 & 0xFF; buff[15] = (rx1 >> 8) & 0x07;
        buff[16] = ry1 & 0xFF; buff[17] = (ry1 >> 8) & 0x03;

        if(write(ramtek, buff, 18) != 18) return (-1);
        return (0);


device_rect_border(x0, y0, x1, y1, color)
int *x0, *y0, *x1, *y1, *color;
{
        register int rx0, ry0, rx1, ry1;

        rx0 = *x0;
        ry0 = *y0;
        rx1 = *x1;
        ry1 = *y1;

        buff[0] = 0x03; buff[1] = 0x0E;
        buff[2] = 0x02; buff[3] = 0x88;
        buff[4] = *color & 0xFF; buff[5] = 0x00;
        buff[6] = alutype; buff[7] = 0x00;
        buff[8] = rx0 & 0xFF; buff[9] = (rx0 >> 8) & 0x07;
        buff[10] = ry0 & 0xFF; buff[11] = (ry0 >> 8) & 0x03;
        buff[12] = 0x10; buff[13] = 0x00;

        buff[14] = rx1 & 0xFF; buff[15] = (rx1 >> 8) & 0x07;
        buff[16] = ry0 & 0xFF; buff[17] = (ry0 >> 8) & 0x03;

        buff[18] = rx1 & 0xFF; buff[19] = (rx1 >> 8) & 0x07;
        buff[20] = ry1 & 0xFF; buff[21] = (ry1 >> 8) & 0x03;

        buff[22] = rx0 & 0xFF; buff[23] = (rx0 >> 8) & 0x07;
        buff[24] = ry1 & 0xFF; buff[25] = (ry1 >> 8) & 0x03;

        buff[26] = rx0 & 0xFF; buff[27] = (rx0 >> 8) & 0x07;
        buff[28] = ry0 & 0xFF; buff[29] = (ry0 >> 8) & 0x03;

        if(write(ramtek, buff, 30) != 30) return (-1);
        return (0);

}

device_rect(x0, y0, x1, y1, color)
int *x0, *y0, *x1, *y1, *color;
{
        register int rx0, ry0, rx1, ry1;

        rx0 = *x0;
        ry0 = *y0;
        rx1 = *x1;
        ry1 = *y1;

        buff[0] = 0x02; buff[1] = 0x09;
        buff[2] = 0x44; buff[3] = 0x08;
        buff[4] = *color & 0xFF; buff[5] = 0x00;
        buff[6] = rx0 & 0xFF; buff[7] = (rx0 >> 8) & 0x07;
        buff[8] = ry0 & 0xFF; buff[9] = (ry0 >> 8) & 0x03;
        buff[10] = rx1 & 0xFF; buff[11] = (rx1 >> 8) & 0x07;
        buff[12] = ry1 & 0xFF; buff[13] = (ry1 >> 8) & 0x03;
        buff[14] = alutype; buff[15] = 0x00;

        if(write(ramtek, buff, 16) != 16) return (-1);
        return (0);
}

device_flipcol(cl)
int *cl;
{
        return(*cl + 125);
}
```

```
device_erase()
{
        int x0, y0, x1, y1, color;

        x0 = y0 = 0;
        x1 = 1279;
        y1 = 1023;
        color = 0;

        return(device_rect(&x0, &y0, &x1, &y1, &color));
}

device_reset()
{
        buff[0] = 0x00; buff[1] = 0x05;

        if(write(ramtek, buff, 2) != 2) return (-1);
        return (0);
}

device_text(x0, y0, fcolor, bcolor, textptr, dx, dy)
int *x0, *y0, *fcolor, *bcolor, *dx, *dy;
char *textptr;
{
        register int wsize, length;
        register char *bptr;
        register int rx0, ry0;

        rx0 = *x0;
        ry0 = *y0;

        buff[0] = 0x0B; buff[1] = 0x0C;
        buff[2] = 0x86; buff[3] = 0x81;
        buff[4] = *fcolor & 0xFF; buff[5] = 0x00;
        buff[6] = *bcolor & 0xFF; buff[7] = 0x00;
        buff[8] = 0x00; buff[9] = 0x00;
        buff[10] = *dx & 0xFF; buff[11] = 0x00;
        buff[12] = *dy & 0xFF; buff[13] = 0x00;
        buff[14] = rx0 & 0xFF; buff[15] = (rx0 >> 8) & 0x07;
        buff[16] = ry0 & 0xFF; buff[17] = (ry0 >> 8) & 0x03;

        length = strlen(textptr);

        if(length <= 0) return(-1);
        if(length > 78) length = 78;  /* Maximum size string which fits buff */

        buff[18] = length & 0xFF; buff[19] = 0x00;
        if((wsize = 20 + length) & 1) wsize++;  /* number of bytes for write */

        bptr = &buff[20];
        while(length--) *bptr++ = *textptr++;

        if(write(ramtek, buff, wsize) < wsize) return(-1);
        return (0);
}

r_wtab(values, load_table, start, nvals, select_table)
char *values;
int load_table, start, nvals, select_table;
{
        register i, j, k;

        j = 0;
        if(nvals) {     /* Then must load before selecting */

        /* Load Auxiliary Memory command -- Device 0 */

                buff[0] = 0x00;
                buff[1] = 0x03;

        /* Table start address -- 16bit start, not entry start */

                buff[2] = (i = start * 2);
                buff[3] = ((i >> 8) + (load_table * 2)) & 0xFF;

        /* Number of bytes to load -- 4 times number of entries */
```

```
                buff[4] = (i = nvals * 4) & 0xFF;
                buff[5] = (i >> 8) & 0xFF;

                j = 6;
                for(i=0;i<nvals;i++) {
                        buff[j++] = *values++;  /* Blue */
                        buff[j++] = *values++;  /* Green */
                        buff[j++] = *values++;  /* Red */
                        buff[j++] = *values++;  /* And any blink... */
                }
        }

        k = j;          /* Either zero or (6 + (nvals * 4)) */

        buff[k++] = 0x00;       /* Select table... */
        buff[k++] = 0x03;

        buff[k++] = 0;          /* Table number */
        buff[k++] = (select_table * 2) & 0x06;

        buff[k++] = 0;          /* Just selecting so no actual writing... */
        buff[k++] = 0;

        if(write(ramtek, buff, k) != k) return(-1);
        return(select_table);
}
```

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10; Lowercase: Yes;-*-
(eval-when (load compile) (load 'mat.o))
; -=========net.l -======
; -=====   At a node -===

(declare (macros t))
(defvar myoutport)
(defvar default-out-file-name '/usr/aic/hota/latest-out.l)
(defvar precision .0001)
(defvar myhash (make-equal-hash-table))
(defvar tobe-updated nil)
(defvar targetnode nil)
(defvar all-nodes nil)
(defvar all-links nil)
(defvar all-nodesh nil)
(defvar all-linksh nil)
(defvar node-w)
(defvar node-h)
(defvar node-w2)
(defvar node-h2)
(defvar node-w4)
(defvar node-h4)
(defvar gp-w 1000)
(defvar gp-h 1000)
(defvar m-x 0)
(defvar m-y 0)
(defvar ma-x)
(defvar device-type nil)
(defvar bgc 0)
(defvar fgc 7)
(defvar hlt-c 1)
(defvar first-c 2)
(defvar second-c 3)
(defvar third-c 4)
(defvar fourth-c 5)
(defvar fifth-c 6)
(defvar need-graphics t)

(defmacro f-dotimes((count count-times) &rest body)
  `(do ((,count 0 (1+ ,count)))
       ((= ,count ,count-times) t)
     ,@body))
(defmacro hash-from-myhash (%lst)
  `(do ((%ans nil (cons (gethash-equal (car %tmp) myhash) %ans))
        (%tmp ,%lst (cdr %tmp)))
       ((null %tmp) (nreverse %ans))))



(mdefflavor node
            ((i-name "noname")
             (values '(True False))
             (rank nil)
             (p-x nil)
             (p-y nil)
             (pos '(0 0))
             (ent 0)
             (imp 0)
             (nd-color 0)
             (tlinks nil)
             (blinks nil)
             (prior nil)
             (init-prior nil)
             (parranks nil)
             (parprobs nil)
             (condpro1 nil)
             (condpro2 nil)
             (belief nil)
             (init-belief nil))
            ()
            :settable-instance-variables)

(mdefflavor link
            ((i-name "noname")
             (tnode nil)
             (bnode nil)
             (indpro nil)
             (lambda nil)
```

```
           (init-lambda nil)
           (pi nil)
           (init-pi nil)
           (t-pt '(0 0))
           (b-pt '(0 0))
           (mu-info 0))
          ()
          :settable-instance-variables)

(defun mdescribe(nnam)
    (describe (gethash-equal nnam myhash)))

(defun shownodes() (mapc 'mdescribe all-nodes) t)
(defun showlinks() (mapc 'mdescribe all-links) t)
(defun shownetwork() (mapc 'mdescribe (append all-nodes all-links)) t)

(defun showbeliefs(&optional (ofwhat all-nodes))
  (cond ((atom ofwhat) (msg (N 1) "belief of " ofwhat " is "
                             (C 35) (msend ofwhat ':belief)))
        (t (mapc (flambda(x)
                          (msg (N 1) " belief of " x " is "
                               (C 35) (msend x ':belief)))
                 ofwhat)))
    t)

(defun showcons(&optional (ofwhat all-links))
  (msg (N 1) (B 30) "pi" (B 10) "lambda")
  (cond ((atom ofwhat)
         (msg (N 1)  ofwhat (C 25) " -->  " (msend ofwhat ':pi) (C 55)
              (msend ofwhat ':lambda)))
        (t (mapc (flambda(x)
                          (msg (N 1)  x (C 25) " -->  " (msend x ':pi) (C 55)
                               (msend x ':lambda)))
                 ofwhat)))
    t)

(defun showlambdas(&optional (ofwhat all-links))
  (cond ((atom ofwhat) (msg (N 1) "lambda of " ofwhat " is "
                             (C 35) (msend ofwhat ':lambda)))
        (t (mapc (flambda(x)
                          (msg (N 1) " lambda of " x " is "
                               (C 35) (msend x ':lambda)))
                 ofwhat)))
    t)

(defun showpis(&optional (ofwhat all-links))
  (cond ((atom ofwhat) (msg (N 1) "pi of " ofwhat " is "
                             (C 35) (msend ofwhat ':pi)))
        (t (mapc (flambda(x)
                          (msg (N 1) " pi of " x " is "
                               (C 35) (msend x ':pi)))
                 ofwhat)))
    t)

(defmethod (node :getallpis) ()
  (cond (tlinks (do ((ans nil
                          (cons (send (gethash-equal
                                        (car tmpl) myhash) ':pi)
                                ans))
                     (tmpl tlinks (cdr tmpl)))
                    ((null tmpl) (nreverse ans))))
        (t (list prior))))

(defmethod (node :getalllambdas) ()
  (cond (blinks (do ((ans nil
                          (cons (send (gethash-equal
                                        (car tmpl) myhash)
                                      ':lambda)
                                ans))
                     (tmpl blinks (cdr tmpl)))
                    ((null tmpl) (nreverse ans))))
        (t (list prior))))


(defmethod (node :update) ()
  (let* ((ptlinksln (length tlinks))
         (pblinksln (length blinks))
         (allpis (send self ':getallpis))
```

```
      (alllambdas (send self ':getalllambdas))
      (piout (norm (outerpro allpis)))
      (prelambda (norm (termpro alllambdas)))
      (bel) (prepi) (contlam))

;; high-light the node
(send self ':high-light t)
(cond ((= ptlinksln 0)                        ; *** no top links ***
       (setq bel (termpro2 prelambda piout)))
      (t
       (setq bel
             (termpro2
               (myfloat 100000000
                        (do ((ans nil
                                  (cons (do ((ansl 0
                                                   (+ ansl
                                                      (* (car mtl)
                                                         (car mt2))))
                                             (mtl (car templ)
                                                  (cdr mtl))
                                             (mt2 temp2 (cdr mt2)))
                                            ((null mtl) ansl))
                                        ans))
                             (templ condpro2 (cdr templ))
                             (temp2 (myfix 10000 piout)))
                            ((null templ) (nreverse ans))))
               prelambda)

             contlam
             (myfloat 100000000
                      (do ((ans nil
                                (cons (do ((ansl 0 (+ ansl
                                                      (* (car mtl)
                                                         (car mt2))))
                                           (mtl (car templ) (cdr mtl))
                                           (mt2 temp2 (cdr mt2)))
                                          ((null mtl) ansl))
                                      ans))
                           (templ condprol (cdr templ))
                           (temp2 (myfix 10000 prelambda)))
                          ((null templ) (nreverse ans)))))))


(let ((oldbel belief))
  (send self ':set-belief (norm bel))       ; update belief
  (send self ':draw-pic oldbel))

(do ((templ blinks (cdr templ))             ; update pis
     (temp2 alllambdas (cdr temp2)))
    ((null templ))
  (msend (car templ) ':draw-pic t)
  (let ((temp3 (mspsend (gethash-equal (car templ)
                                        myhash)
                        ':pi (norm (matdiv bel (car temp2))))))
    (cond (temp3 (push temp3 tobe-updated))))
  (msend (car templ) ':draw-pic))

(cond ((= ptlinksln 1)                      ; update lambdas
       (msend (car tlinks) ':draw-pic t)
       (let ((temp (mspsend (gethash-equal (car tlinks) myhash)
                            ':lambda (norm contlam))))
         (cond (temp (push temp tobe-updated))))
       (msend (car tlinks) ':draw-pic))
      ((> ptlinksln 1)
       (do ((templ tlinks (cdr templ))
            (temp2 (maklis (length parranks)) (cdr temp2))
            (temp3 (termpro2 contlam piout)))
           ((null templ))
         (msend (car templ) ':draw-pic t)
         (let* ((tclh (gethash-equal (car templ)
                                      myhash))
                (temp (mspsend tclh ':lambda
                               (norm
                                 (arrange temp3
                                          parranks (car temp2)
                                          (send tclh ':pi))))))
           (cond (temp (push temp tobe-updated))))
         (msend (car templ) ':draw-pic))))
    ;; de-high-light the node
    (send self ':high-light nil)))


; -=*-=*-=*

(defmethod (node :find-ys)()
  "  finds the y-co-ordinate (relative) of the node in the network"
  (cond (p-y)
        (t (let ((prnts (do ((ans nil
                                  (cons (gethash-equal
                                          (msend (car temp) ':tnode) myhash)
                                        ans))
                             (temp tlinks (cdr temp)))
                            ((null temp) ans))))
             (send self ':set-p-y
                   (cond ((null prnts) 1)
                         (t (+ 1
                               (do ((ans (send (car prnts) ':find-ys)
                                         (max ans (send (car temp) ':find-ys)))
                                    (temp (cdr prnts) (cdr temp)))
                                   ((null temp) ans))
                               ))))))))

(defun find-xys()
  "  finds x,y positional co-ordinates (relative) of all nodes
     in the network and save them as p-x and p-y in the node. "
  (setq m-y 0 m-x 0)                                ;re-initializing
  (for-each j all-nodesh                            ;re-initializing
            (send j ':set-p-x nil) (send j ':set-p-y nil))
  (for-each j all-nodesh
            (setq m-y (max m-y (send j ':find-ys))))
  (setf ma-x (array nil t m-y 2))
  (let ((pre (do ((ans nil (cond ((= m-y (msend (car tmp) ':p-y))
                                  (cons (car tmp) ans))
                                 (t ans)))
                  (tmp all-nodes (cdr tmp)))
                 ((null tmp) (nreverse ans))))
        prel preh)
    (setq prel (length pre)
          preh (hash-from-myhash pre))
    (f-dotimes
      (jk m-y)
      (setf (aref ma-x (- m-y 1 jk) 0) prel)
      (setf m-x (max m-x prel))
      ;; set p-x and p-y for the pre nodes
      (do ((temp preh (cdr temp))
           (tempc 1 (1+ tempc)))
          ((null temp) t)
        (send (car temp) ':set-p-x tempc)
        (send (car temp) ':set-p-y (- m-y jk)))
      [illegible] pre (remove-duplicates
                (do ((ans nil
                          (cons (msend (car temp) ':tnode) ans))
                     (temp (do ((ans1 nil
                                      (append ans1
                                              (send (car templ) ':tlinks)))
                                (templ preh (cdr templ)))
                               ((null templ) ans1))
                           (cdr temp)))
                    ((null temp) (nreverse ans)))
                t)
            prel (length pre)
            preh (hash-from-myhash pre)))))

(defun find-pos()
  "  determinies the size of each node ie. node-width and height &
     finds the absolute co-ordinates of each node in the network
     (for the display window) and saves them in pos slot of each node.
     Also finds the absolute co-ordinates of each link in the network
     and saves them T-PT(top point)  & B-PT(bottom point) of each link. "
  (setq gp-w (graph-pane-width)                     ;graphic pane width
        gp-h (graph-pane-height)                    ;graphic pane height
        node-w (fix (/$ gp-w 1.5 m-x))
        node-h (fix (/$ gp-h 1.5 m-y)))
  (or (evenp node-w) (incf node-w))
  (or (evenp node-h) (incf node-h))
  (setq node-w2 (/ node-w 2) node-h2 (/ node-h 2)
             node-w4 (/ node-w2 2) node-h4 (/ node-h2 2))
        (f-dotimes (x m-y)
                   (setf (aref ma-x x 1)
                         (+ node-w4 node-w2
                            (fix (*$ (- m-x (aref ma-x x 0)) 0.75 node-w)))))

        (for-each xh all-nodesh                         ; node co-ordinates
                  (let ((t-x (send xh ':p-x))
                        (t-y (send xh ':p-y)))
                    (send xh ':set-pos
                          (list (+ (aref ma-x (1- t-y) 1)
                                   (* (1- t-x) (+ node-w node-w2)))
                                (+ node-h2 node-h4 (* (1- t-y)
                                                      (+ node-h node-h2)))))))
        (for-each xh all-linksh                         ; link co-ordinates
                  (let ((y1 (msend (send xh ':tnode) ':pos))
                        (y2 (msend (send xh ':bnode) ':pos))
                        x1 x2 slope a1 b1 a2 b2)
                    (setq x1 (car y1) y1 (cadr y1)
                          x2 (car y2) y2 (cadr y2))
                    (cond ((= x1 x2)
                           (setq a1 x1 a2 x1 b1 (+ y1 node-h2) b2 (- y2 node-h2)))
                          ((> x1 x2)
                           (setq slope (/$ (float (- y2 y1)) (- x2 x1))
                                 a1 (max (- x1 node-w2)
                                         (fix (+$ x1 (/$ node-h2 slope))))
                                 b1 (min (+ y1 node-h2)
                                         (fix (-$ y1 (*$ node-w2 slope) -0.999)))
                                 a2 (min (fix (-$ x2 (/$ node-h2 slope) -0.999))
                                         (+ x2 node-w2))
                                 b2 (max (- y2 node-h2)
                                         (fix (+$ y2 (*$ node-w2 slope))))))
                          (t (setq slope (/$ (float (- y2 y1)) (- x2 x1))
                                   a1 (min (fix (+$ x1 (/$ node-h2 slope) 0.999))
                                           (+ x1 node-w2))
                                   b1 (min (+ y1 node-h2)
                                           (fix (+$ y1 (*$ node-w2 slope) 0.999)))
                                   a2 (max (- x2 node-w2)
                                           (fix (-$ x2 (/$ node-h2 slope))))
                                   b2 (max (- y2 node-h2)
                                           (fix (-$ y2 (*$ node-w2 slope)))))))
                    (send xh ':set-t-pt (list a1 b1))
                    (send xh ':set-b-pt (list a2 b2)))))

(defmethod (node :high-light) (&optional (flg t))
  (let ((x1 (- (car pos) node-w2))
        (x2 (+ (car pos) node-w2))
        (y1 (+ 10 (- (cadr pos) node-h2)))
        (y2 (+ (cadr pos) node-h2)))
    (draw-myrectangle-border
      x1 y1 x2 y2 (cond (flg (+ 8 hlt-c)) (t fgc)))))

(defmethod (node :draw-pic) (&optional (flg nil))
  "  draws the node and its beliefs as a histogram in the display
     at the proper place (ie. at the value of POS of that node."
  (let ((x1 (- (car pos) node-w2))
        (x2 (+ (car pos) node-w2))
        (y1 (+ 14 (- (cadr pos) node-h2)))
        (y2 (+ (cadr pos) node-h2))
        (ng-h (- node-h 15))
        (ng-w (/ node-w (+ rank rank 1))))
    (cond (flg
           (do ((tmp1 belief (cdr tmp1))
                (tmp2)
                (tmp3 flg (cdr tmp3))
                (tmp4)
                (d-x1 (+ x1 (/ (- node-w (* ng-w (+ rank rank -1))) 2))
                      (+ d-x1 ng-w ng-w)))
               ((null tmp1) t)
             (setq tmp2 (fix (*$ ng-h (car tmp1)))
                   tmp4 (fix (*$ ng-h (car tmp3))))
             (cond ((= tmp2 tmp4))
                   ((> tmp2 tmp4)
                    (draw-myrectangle ng-w (- tmp2 tmp4)
                                      d-x1 (- y2 tmp2) fgc))
                   ((< tmp2 tmp4)
                    (draw-myrectangle ng-w (- tmp4 tmp2)
                                      d-x1 (- y2 tmp4) bgc))))
```

```
            (t
             (draw-myrectangle node-w 10 x1 (- y1 13) fgc)
             (cond ((= nd-color 0)
                    (draw-myline x2 y1 x2 y2 fgc)
                    (draw-myline x2 y2 x1 y2 fgc)
                    (draw-myline x1 y2 x1 y1 fgc))
                   (t (draw-myrectangle (- x2 x1) (- y2 y1) x1 y1 nd-color)))
             (draw-mystring i-name (+ x1 2) (- y1 13)
                            (- x2 2) (- y1 1) bgc fgc)
             (do ((tmp1 belief (cdr tmp1))
                  (tmp2)
                  (d-x1 (+ x1 (/ (- node-w (* ng-w (+ rank rank -1))) 2))
                        (+ d-x1 ng-w ng-w)))
                 ((null tmp1) t)
               (setq tmp2 (fix (*$ ng-h (car tmp1))))
               (draw-myrectangle ng-w tmp2
                                 d-x1 (- y2 tmp2) fgc))))))

(defmethod (link :draw-pic) (&optional (flg nil))
  "  draws a line representing the link on the display area between
     T-PT and B-PT of that link."
  (draw-myline (car t-pt) (cadr t-pt) (car b-pt) (cadr b-pt)
               (cond (flg (+ 8 hlt-c)) (t fgc))))

(defmethod (node :change-bg-color) ()
  "  draws the node and its beliefs as a histogram in the display
     at the proper place (ie. at the value of POS of that node."
  (let ((x1 (- (car pos) node-w2))
        (x2 (+ (car pos) node-w2))
        (y1 (+ 10 (- (cadr pos) node-h2)))
        (y2 (+ (cadr pos) node-h2))
        (ng-h (- node-h 12))
        (ng-w (/ node-w (+ rank rank 1))))

    (draw-myrectangle (- x2 x1) (- y2 y1) x1 y1 nd-color)
    (draw-myline x2 y1 x2 y2 fgc)
    (draw-myline x2 y2 x1 y2 fgc)
    (draw-myline x1 y2 x1 y1 fgc)

    (do ((tmp1 belief (cdr tmp1))
         (tmp2)
         (d-x1 (+ x1 (/ (- node-w (* ng-w (+ rank rank -1))) 2))
               (+ d-x1 ng-w ng-w)))
        ((null tmp1) t)
      (setq tmp2 (fix (*$ ng-h (car tmp1))))
      (draw-myrectangle ng-w tmp2
                        d-x1 (- y2 tmp2) fgc))))

(defun drawpic(&optional (fg nil))
  "  draws the nodes and links on the display area."
  (and fg (clear-graph-pane))
  (for-each j (append all-nodesh all-linksh)
      (send j ':draw-pic))
  t)

(defmethod (link :cal-mu-info)()
  "  finds the mutual-information of a link "
  (send self ':set-mu-info
        (do ((ans 0 (+$ ans
                        (do ((ans1 0 (+$ ans1
                                         (cond ((zerop (car tmp4)) 0)
                                               ((or (zerop (car tmp1))
                                                    (zerop (car tmp2))) 25.0)
                                               (t
                                                (*$ (car tmp4)
                                                    (log (/$ (car tmp4)
                                                             (car tmp1)
                                                             (car tmp2)))))))))
                             (tmp2 tmp5 (cdr tmp2))
                             (tmp4 (car tmp3) (cdr tmp4)))
                            ((null tmp4) ans1))))
             (tmp1 (msend bnode ':belief)
                   (cdr tmp1))
             (tmp3 indprc
                   (cdr tmp3))
             (tmp5 (msend tnode ':belief)))
            ((null tmp1) ans))))
```

```
(defmethod (node :cal-imp)()
  (let ((sumtop (do ((ans 0 (+$ ans (msend (car temp) ':mu-info)))
                     (temp tlinks (cdr temp)))
                    ((null temp) ans)))
        (sumbot (do ((ans 0 (+$ ans (msend (car temp) ':mu-info)))
                     (temp blinks (cdr temp)))
                    ((null temp) ans)))
        newnodesh tmph)
    (for-each x (hash-from-myhash tlinks)
              (setq tmph (gethash (send x ':tnode) myhash))
              (cond ((send tmph ':imp) nil)
                    (t (send tmph ':set-imp
                             (/$ (*$ imp (send x ':mu-info)) sumtop))
                       (push tmph newnodesh))))
    (for-each x (hash-from-myhash blinks)
              (setq tmph (gethash (send x ':bnode) myhash))
              (cond ((send tmph ':imp) nil)
                    (t (send tmph ':set-imp
                             (/$ (*$ imp (send x ':mu-info)) sumbot))
                       (push tmph newnodesh))))
    (for-each y newnodesh (send y ':cal-imp))))

(defun find-importance()
  "  finds the mutual information of each link and then finds the
     importance factors of each node in the network."
  ;; first find mutual information of each link
  (for-each x all-linksh (send x ':cal-mu-info))
  ;; then find the importance factors of each node
  (or (gethash targetnode myhash)
      (set-target-node 1))
  (for-each x all-nodesh (send x ':set-imp nil))
  (msend targetnode ':set-imp 1.0)
  (msend targetnode ':cal-imp))

(defmethod (node :cal-ent)()                     ;not needed for the time being
  "  finds the entropy of a node and saves it in ENT of the node."
  (send self ':set-ent
        (do ((ans 0 (+$ ans
                        (cond ((or (zerop (car tmp1))
                                   (= (car tmp1) 1.0)) 0)
                              (t (* (car tmp1)
                                    (log (car tmp1)))))))
             (tmp1  belief (cdr tmp1)))
            ((null tmp1) ans))))

(defun find-entropy()
  "  finds the entropy of all nodes and saves them in the ENT slot
    of each node."
  (for-each x all-nodesh (send x ':cal-ent)))


(defun set-gray()                               ; this is w.r.t importance
  "  grays all nodes in the network, the intensity depending on each
     nodes importance in the network with respect to the target node."
  (for-each j all-nodesh (send j ':set-nd-color 0))     ;initializing
  (let* ((avg-imp 0)
         (temp (do ((ans nil
                         (cons (list (send (car tmp) ':imp) (car tmp))
                               ans))
                    (tmp ;;(remove (gethash-equal targetnode myhash) all-nodesh)
                      (hash-from-myhash (remove targetnode all-nodes))
                      (cdr tmp)))
                   ((null tmp) ans)
                 (incf-f avg-imp (send (car tmp) ':imp)))))
    (setq avg-imp (/$ avg-imp (length temp))
          temp (sortcar temp #'>))
    (do ((tmp1 temp (cdr tmp1))
         (tmp2 (list first-c second-c third-c fourth-c fifth-c)
               (cdr tmp2)))
        ((or (null tmp1) (null tmp2)) t)
      (or (< (caar tmp1) avg-imp)
          (send (cadar tmp1) ':set-nd-color (car tmp2))))))

(defun init-net()
  ;; add tlinks and blinks to nodes & check cardinality of indpro
  (for-each xh all-nodesh
            (let ((pri (send xh ':prior)))
              (send xh ':set-rank (length (send xh ':values)))
```

```
               (cond (pri (send xh ':set-prior (norm pri)))
                     (t (send xh ':set-prior
                              (listsomany (send xh ':rank)
                                  (/$ 1.0 (send xh ':rank))))))
               (send xh ':set-belief (send xh ':prior))))

  (for-each x all-links
            (let ((xh (gethash-equal x myhash)))
               (msendal (gethash-equal (send xh ':tnode) myhash)
                        ':set-blinks ':blinks x)
               (msendal (gethash-equal (send xh ':bnode) myhash)
                        ':set-tlinks ':tlinks x)))


                                                 ; expand all nodes
  (for-each xh all-nodesh
            (let ((tlk (send xh ':tlinks)))
               (cond (tlk (do ((parpro nil
                                       (cons (send (gethash-equal (car tlks)
                                                                  myhash) ':indpro)
                                             parpro))
                               (parrnk nil
                                       (cons (msend (msend (car tlks) ':tnode)
                                                    ':rank)
                                             parrnk))
                               (tlks (reverse tlk) (cdr tlks)))
                              ((null tlks)
                               (send xh ':set-parprobs parpro)
                               (send xh ':set-parranks parrnk)
                               (send xh ':set-condpro1
                                      (findcp parpro))
                               (send xh ':set-condpro2
                                     (invert (send xh ':condpro1)))
                               ))))))

                                                 ; expand all links
  (for-each xh all-linksh
            (send xh ':set-pi
                  (norm (send (gethash-equal (send xh ':tnode)
                                                    myhash) ':prior)))
            (send xh ':set-lambda
                  (norm
                  (do ((ans nil
                            (cons (do ((ans1 0
                                             (+$ ans1 (*$ (car mt1)
                                                          (car mt2))))
                                       (mt1 (car tem1) (cdr mt1))
                                       (mt2 tem2 (cdr mt2)))
                                      ((null mt1) ans1))
                                  ans))
                       (tem1 (invert (send xh ':indpro)) (cdr tem1))
                       (tem2 (send (gethash-equal (send xh ':bnode) myhash)
                                   ':prior)))
                      ((null tem1) (nreverse ans))))))
  (find-xys)
  (find-pos)
  t)

(defun d-l-node (x)
  (let ((xh (gethash-equal x myhash)))
    (msg (P myoutport)
         (N 1) "(mmake-instance '" x " 'node"
         (N 1) " ':i-name      \"" (send xh ':i-name) "\""
         (N 1) " ':values      '" (send xh ':values)
         (N 1) " ':rank        '" (send xh ':rank)
         (N 1) " ':p-x         '" (send xh ':p-x)
         (N 1) " ':p-y         '" (send xh ':p-y)
         (N 1) " ':pos         '" (send xh ':pos)
         (N 1) " ':tlinks      '" (send xh ':tlinks)
         (N 1) " ':blinks      '" (send xh ':blinks)
         (N 1) " ':prior       '" (send xh ':prior)
         (N 1) " ':parranks    '" (send xh ':parranks)
         (N 1) " ':parprobs    '" (send xh ':parprobs)
         (N 1) " ':condpro1    '" (send xh ':condpro1)
         (N 1) " ':condpro2    '" (send xh ':condpro2)
         (N 1) " ':belief      '" (send xh ':belief) ")"
         (N 2))))
```

```
(defun d-l-link (x)
  (let ((xh (gethash-equal x myhash)))
    (msg (P myoutport)
         (N 1) "(mmake-instance '" x " 'link"
         (N 1) " ':i-name     \"" (send xh ':i-name) "\""
         (N 1) " ':tnode      '" (send xh ':tnode)
         (N 1) " ':t-pt       '" (send xh ':t-pt)
         (N 1) " ':bnode      '" (send xh ':bnode)
         (N 1) " ':b-pt       '" (send xh ':b-pt)
         (N 1) " ':indpro     '" (send xh ':indpro)
         (N 1) " ':lambda     '" (send xh ':lambda)
         (N 1) " ':pi         '" (send xh ':pi) ")"
         (N 2))))

(defun down-load-all()
  "  writes the declarations, values of globals and  node's and link's
     information (actually a describe) onto a file so it can be loaded
     (future use) as it is and run the program without initialization
     of the network which is time consuming."
  (msg (N 1) " Name of the output file to write : ")
  (setf myoutport (outfile (cond ((read))
                                 (t default-out-file-name))))
  (msg (P myoutport)
       (N 1) "(defvar gp-w)"
       (N 1) "(defvar gp-h)"
       (N 1) "(defvar node-w)"
       (N 1) "(defvar node-h)"
       (N 1) "(defvar node-w2)"
       (N 1) "(defvar node-h2)"
       (N 1) "(defvar node-w4)"
       (N 1) "(defvar node-h4)"
       (N 1) "(defvar m-x)"
       (N 1) "(defvar m-y)"

       (N 1) "(setq gp-w " gp-w
       (N 1) "      gp-h " gp-h
       (N 1) "      node-w " node-w
       (N 1) "      node-h " node-h
       (N 1) "      node-w2 " node-w2
       (N 1) "      node-h2 " node-h2
       (N 1) "      node-w4 " node-w4
       (N 1) "      node-h4 " node-h4
       (N 1) "      m-x " m-x
       (N 1) "      m-y " m-y ")")

  (msg (P myoutport) (N 1) "; -=-=-=-=-=NODES-=-=-=-=" (N 1))
  (mapc 'd-l-node (get-insts 'node))
  (msg (P myoutport) (N 1) "; -=-=-=-=-=LINKS-=-=-=-=" (N 1))
  (mapc 'd-l-link (get-insts 'link)))

(defun copy-info()
  (for-each x all-nodesh
            (send x ':set-init-prior (send x ':prior))
            (send x ':set-init-belief (send x ':belief)))
  (for-each x all-linksh
            (send x ':set-init-lambda (send x ':lambda))
            (send x ':set-init-pi (send x ':pi))))

(defun mreset()                               ; reset the network
  (let ((old-myhash myhash))
    (setq myhash (make-equal-hash-table))       ; create new hash table
    (puthash '(node +info) (gethash '(node +info) old-myhash) myhash)
    (puthash '(link +info) (gethash '(link +info) old-myhash) myhash)
    (msend '(node +info) ':set-insts nil)
    (msend '(link +info) ':set-insts nil)
    (setq all-nodes nil all-links nil all-nodesh nil all-linksh nil)))

(defun show-dependence()
  (find-importance)
  (set-gray)
  (drawpic 1))

(defun dependency()
  (set-target-node)
  (show-dependence))

(defun set-target-node(&rest ignore)
  (let (ttarg )
    (msg (N 1) " Target node : ")
    (setq ttarg (read))
    (cond ((gethash-equal ttarg myhash) (setq targetnode ttarg))
          (t (set-target-node)))))
;;; ========== TOP level functions ============

(eval-when (laod compile eval)
  (setq precision .0001 float-format "%.4g"))
(setq float-format "%.4g")
(defun load-init(&optional (timeflag nil))
  (mreset)
  (and timeflag (msg (N 1) (matdiv (ptime) '(60 60)) (N 1)))
  (msg (N 1) " Name of the input file to load : ")
  (load  (read))
  (setq all-nodes (get-insts 'node)
        all-links (get-insts 'link)
        all-nodesh (do ((ans nil
                             (cons (gethash-equal (car tmp) myhash)
                                   ans))
                        (tmp all-nodes (cdr tmp)))
                       ((null tmp) (nreverse ans)))
        all-linksh (do ((ans nil
                             (cons (gethash-equal (car tmp) myhash)
                                   ans))
                        (tmp all-links (cdr tmp)))
                       ((null tmp) (nreverse ans))))

  (msg (N 1) " Initialize ? ")
  (and (ttyesno) (init-net) (down-load-all))
  (and timeflag (msg (N 1) (matdiv (ptime) '(60 60)) (N 1)))
  (drawpic 1)
  (updateall-br)
  (and timeflag (msg (N 1) (matdiv (ptime) '(60 60)) (N 1)))
  (copy-info))

(defun reset-net()
  (for-each x all-nodesh
            (send x ':set-prior (send x ':init-prior))
            (send x ':set-belief (send x ':init-belief)))
  (for-each x all-linksh
            (send x ':set-lambda (send x ':init-lambda))
            (send x ':set-pi (send x ':init-pi)))
  (show-dependence))


(defun updateall-br(&optional (what-node nil))
  (for-each x all-nodesh                        ;remove back-ground colors
            (cond ((= (send x ':nd-color) bgc))
                  (t (send x ':set-nd-color bgc)
                     (send x ':change-bg-color))))
  (let ((tobe-updated (cond (what-node (list what-node))
                            (t all-nodes)))
        (current))
    (while tobe-updated
           (setq tobe-updated (remove-duplicates tobe-updated)
                 current (gethash-equal (car (last tobe-updated))
                                        myhash)
                 tobe-updated (reverse (cdr (reverse tobe-updated))))
           (send current ':update)))
  (find-importance)
  (set-gray)
  (for-each x all-nodesh                        ;set colors
            (or (= (send x ':nd-color) bgc)
                (send x ':change-bg-color))))


;;; ====updateall-dn====
;;;             (setq current (gethash-equal (car tobe-updated) myhash)
;;;                   tobe-updated (cdr tobe-updated))
;;;             (send current ':update)))

;;; ====updateall-bn====
;;;             (setq current (gethash-equal (car (last tobe-updated))
;;;                                        myhash)
;;;                   tobe-updated (reverse (cdr (reverse tobe-updated))))

;;; ====updateall-dp====
;;;             (setq tobe-updated (remove-duplicates tobe-updated)
;;;                    current (gethash-equal (car tobe-updated) myhash)
;;;                    tobe-updated (cdr tobe-update- '

;;; ====updateall-dpe====
;;;              (setq tobe-updated (remove-duplicates tobe-updated t)
;;;                    current (gethash-equal (car tobe-updated) myhash)
;;;                    tobe-updated (cdr tobe-updated))

;;; ====updateall-bre====
;;;              (setq tobe-updated (remove-duplicates tobe-updated t)
;;;                    current (gethash-equal (car (last tobe-updated))
;;;                                myhash)
;;;                    tobe-updated (reverse (cdr (reverse tobe-updated))))

(defun change-effects(&optional
                (leaf-nodes
                  (do ((ans nil (cond ((null (msend (car temp) ':blinks))
                                       (cons (car temp) ans))
                                      (t ans)))
                       (temp all-nodes (cdr temp)))
                      ((null temp) ans))))
  (and (atom leaf-nodes) (setf leaf-nodes (list leaf-nodes)))
  (for-each x leaf-nodes (get-new-val x))
  (updateall-br)
  (or (member targetnode all-nodes) (set-target-node))
  (and targetnode
       (do ((tnval (msend targetnode ':values) (cdr tnval))
            (tnbel (msend targetnode ':belief) (cdr tnbel)))
           ((null tnval) t)
         (msg (N 1) "  " (car tnval) (C 20) " --> " (car tnbel)))))

(defun change-causes(&optional
                (top-nodes
                  (do ((ans nil (cond ((null (msend (car temp) ':tlinks))
                                       (cons (car temp) ans))
                                      (t ans)))
                       (temp all-nodes (cdr temp)))
                      ((null temp) ans))))
  (and (atom top-nodes) (setf top-nodes (list top-nodes)))
  (for-each x top-nodes (get-new-val x))
  (updateall-br)
  (or (member targetnode all-nodes) (set-target-node))
  (and targetnode
       (do ((tnval (msend targetnode ':values) (cdr tnval))
            (tnbel (msend targetnode ':belief) (cdr tnbel)))
           ((null tnval) t)
         (msg (N 1) "  " (car tnval) (C 20) " --> " (car tnbel)))))

(defun get-new-val(x)
  (prog(xx)

       (msg (N 1) "  " x "s values are :" (msend x ':values))
       (msg (N 1) "      prior values: " (msend x ':prior))
       lop
       (msg (N 1) "    Enter new evidence : ")
       (setq xx (read))
       (cond ((null xx))
             ((= (length xx) (msend x ':rank))
              (msend x ':set-prior (norm xx)))
             (t (msg (N 1) " Error  **wrong cardinality**. Try again.")
                (go lop)))))
```